Alexandru Ioan Cuza
University of Iasi

Malware Detection and Analysis

PhD thesis summary

Author:
Catalin Valeriu
Lita

Supervisor:
Prof. Dr. Ferucio
Laurentiu Tiplea

2018

List of published research papers

1. Catalin-Valeriu Lita. “On Complexity of the Detection Problem for Bounded Length Polymorphic Viruses”. In: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2016 18th International Symposium on. IEEE. 2016, pp. 371–378.

2. Catalin Valeriu Lita, Doina Cosovan, and Dragos Gavrilut. "Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in UPA packers”. In: Journal of Computer Virology and Hacking Techniques 14.2 (2018), pp. 107–126.

3. Doina Cosovan and Catalin Valeriu Lita. “Practical aspects related to using Hidden Markov Models for detecting metamorphic file infectors”. In: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). 2017. (to appear)

Presentations at CARO workshop

CARO workshop is specific to Anti-Malware industry and press is not allowed to attend. It is with closed doors and only the abstracts are made public on their website.

1. Catalin Valeriu Lita and Doina Cosovan. "An Incursion in the Malware Packer Market”. CARO 2017, Krakow, Poland.

   https://www.eiseverywhere.com/ehome/caro2017/

2. Catalin Valeriu Lita and Doina Cosovan. "Evading Detection with Anti-Emulation Techniques”. CARO 2018, Portland, United States of America.

   https://caroworkshop2018.wordpress.com/

Contents

Chapter 1

Introduction

In 1984 the term computer virus was defined by Fred Cohen as "a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself."[Coh87]. Computer viruses were considered by that time as "a major computer security problem"[Coh87], but only a theoretical one as real computer viruses were not yet present.

One of the first computer viruses is Brain virus, known to be released on January 1986. This virus infected the boot sector of floppy disks [FS]. Its intentions were to protect a medical software from piracy.

As new computer viruses were created not long after the release of Brain virus, real security risks emerged from their presence as real viruses. As a consequence, soon was a need for security products, known as antivirus or Anti-Malware products, to protect computers from viruses. An antivirus product is a software application that protects a computer from future infections and removes the infections if the computer is already infected. To solve this problem, it first has to detect the virus code in an infected system, known as the detection problem for computer viruses.

At the beginning, detecting computer viruses was not complicated, the usage of simple pattern matching was enough. Later, computer viruses evolved to use specific methods to evade detection.

One of the most known methods to evade detection is polymorphism. With polymorphism different instances of computer virus code have the same behavior on execution, but their form can be different and simple pattern matching is not enough to detect them. A polymorphic engine is the virus component that rewrites a sequence of instructions to another sequence that has the same behavior on execution as the original sequence but with a different form.

As computer viruses continued to evolve in the past 30 years, one of the main feature that also evolved is the evasion of antivirus detection. Antivirus products and viruses keep updating each other, viruses to evade detection and antiviruses to add detection to the new updated and undetected viruses. This is why we consider that the detection problem for computer viruses is of high importance for academic and industry research.

To be able to compute the complexity for the detection problem of polymorphic viruses we used grammar based formalisms. This way we associate a grammar with a polymorphic engine and the rewrite rules of a specific grammar are associated with the rewrite rules of the polymorphic engine. Thus, the complexity for the detection problem of the instances generated by a specific polymorphic engine will be equivalent with the recognition problem for a fixed grammar

G, mainly the problem to decide if given a string w, whether $w \in L(G)$.

A malware program is a more generic term that also includes viruses as were defined in 1984. The term virus is restricted to malware programs that infect other programs with their code. There are also malware programs with malicious behavior that execute without having this specific infection behavior. A malware program is defined more general as a program that has a malicious behavior, for example allowing remote access to infected host for malware creators, or stealing e-mail credentials.

Malware creators work on creating malware programs that have a specific malicious objective. After the malware program is propagated to enough computers, soon, Anti-Malware products will get that malware sample and add a detection for it. Next time the malware creator tries to propagate that malware program to a new computer that has an Anti-Malware product installed, it will be detected after a specific a detection was added for it. Now, that malware sample is detected, removed from current infected systems and unable to propagate to new systems that are protected by the Anti-Malware product. As this can happen to any malicious program, malware creators had to update their malware programs in a way that will not be detected by Anti-Malware programs before spreading again. One of the methods of evading detection was to add another layer of protection on their original malicious code. For example encrypting all the original code and decrypting it on run-time before executing it. In this way the original malicious code was not easily accessible to be detected, only the code that decrypts and executes it. As an additional layer of protection, the code that is responsible for the decryption and execution of the payload will be obfuscated and generated with a polymorphic engine. In the case of viruses, those that spread by infecting other programs, they have to contain the polymorphic engine in their code so it can be used when generating new instances. Because anyone that wanted to spread malware programs wanted for his samples to be undetected by Anti-Malware products, a new type of programs evolved, known as malware packers that are adding another layer of protection on the original malware program similar with what malware creators had to do to evade detection, but this time it is a dedicated program that will handle only the objective of evading detection and usually will be available as a service.

A packer, also named protector, was known as an application that protects commercial software from anti-reversing. With a similar principle, a malware packer is a protector for a malware file that keeps it undetected for a period of time from Anti-Malware products. To achieve this objective, malware packers use techniques like polymorphism, obfuscation, anti-debugging, anti-disassembly, anti-emulation.

Because polymorphism can be bypassed generically with an emulator, anti-emulation techniques were added as a way to evade detection after Anti-Malware products started to implement and update its emulators.

An **emulator** simulates an operating system, with its own memory, disk, registry and processes. In this emulated operating system a malware can be executed without affecting the guest operating system. In our case the emulator is used as a tool for detecting malicious samples based on their behavior and artifacts obtained in the emulated system. This emulator does not simulate a perfect real operating system, just a minimal one. That is because it has some constraints:

- it must not use too much space on disk or in memory (100MB on disk sometimes can be considered too much)
- it can not contain all files found in a real operating system, not even all from system32 directory
- the content of system files from an emulator is not identical to the ones from a real operating system, because there is a minimal implementation

- it does not have implementation for all Windows API functions

- for some Windows API functions, even if it has an implementation, is not a perfect one, for same parameters it could return something different compared to what is returned by the original API function

- it must be fast, not to take minutes to run a sample in an emulator, for that reason it can not run instructions infinitely, some limits are added for the execution time

- it must run on any processor architecture, for example on ARM or on AMD64.

An **anti emulation technique** is used to detect or stop the execution when the program is running in an emulator and not in a real operating system. Once it knows it is not executing in a real operating system, it can just execute different code, usually it stops execution faster or enters in an infinite loop when it detects that it is running in an emulator.

Anti-emulation techniques are used by most malware packers to evade Anti-Malware programs detection. Still, this subject did not get big attention in research papers, mainly because it is a restricted domain, well known only for those who work at Anti-Malware products or the malware packer creators themselves.

This theses has 3 parts, first part is about complexity results for bounded length viruses where we used formal grammars, the second part is about anti-emulation techniques used by malware packers and the last part is about using Hidden Markov Models to detect metamorphic code.

Chapter 2

Complexity results for bounded length polymorphic viruses

Cohen proposed the first formal models for viruses in 1984 [Coh85] using Turing machines. Other formalisms followed, using recursive functions [BKM07; Adl90] and formal grammars [Fil07; Qoz99].

Cohen [Coh85] showed that detection problem is undecidable in the general case for a viral set. In case of some restrictions, like bounded length, the detection problem can be no more undecidable and its complexity is important. Thus, in 2003, Spinellis gave examples of bounded length polymorphic viruses that solve Boolean satisfiability problem and for which the detection complexity is NP-complete [Spi03].

This thesis chapter focuses on the reliable detection problem for bounded length polymorphic viruses. Formal grammars are used as a modeling formalism. An example of a grammar for which the recognition problem is PSPACE-complete is given and using that particular grammar an example of a PSPACE-complete virus is given [Lit16].

The recognition problem for a fixed grammar G is important in our research because it will be used to obtain complexity results for specific computer viruses that correspond to specific types of formal grammars. The Recognition Problem for a fixed grammar G ($RP(G)$) is defined as the problem to decide, for a given a string w, whether $w \in L(G)$. In this case the grammar G is a fixed grammar known from the start.

Next we give example of a context sensitive grammar for which the recognition problem is PSPACE-complete as we didn't find an example in known research literature.

We start by presenting the QSAT problem.

$X = \{x_1, \ldots, x_n\}$ is a set of boolean variables. A QSAT formula has the following form:

$$\varphi = (Q^1 x_1) \cdots (Q^n x_n) C_1 \wedge \cdots \wedge C_m$$

In this formula C_i is called a *clause.* A *clause* is a disjunction of variables, for example $x_5 \vee \bar{x}_0 \vee x_1$. Q^i is a quantifier that can be one of $\forall$ or $\exists$.

The QSAT problem has to decide if φ formula is true or false. To do that it has to evaluate lots of assignments. In the case that all the quantifiers are $\exists$ the QSAT problem is reduced to Satisfiability problem that is known to be NP-complete.

The next step is to encode a QSAT formula to our grammar symbols. For the following particular example of a QSAT formula,

$$\varphi = (\exists x_1)(\forall x_2)(\forall x_3)(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$$

we compute the fallowing encoding:

$$\exists_{\_,\_,0}\forall_{\_,\_,0}\forall_{\_,\_,0}(xaxbbxaaa)(xbxaaxbbb)$$

Those 3 indexes that we have for the quantifiers will be used later for evaluating the formula.

Because we had a considerable number of rules, we grouped them by their functionality in 6 sets of rules:

- R0 - generating QSAT formula to evaluate
- R1 - propagate current assignment to clauses
- R2 - evaluating the truth value for current assignment
- R3 - propagating previous result
- R4 - generate next assignment in lexicographic order
- R5 - obtain the final result after all assignments are evaluated

Our final PSPACE-complete grammar has the following form:

- $G = (V,\, \Sigma,\, R,\, S)$
- $V = \{a,\, b,\, x,\, (,\,),\, S,\, S_1,\, S_2,\, X,\, A,\, B,\, S_3,\, S_4,\, S_{4,0},\, S_{4,1},\, S_{5,0},\, S_{5,1},\, S_{6,0},\, S_{6,1},\, S_6,\, b_1,\, b_0,\, a_0,\, a_1,\, S_{exit},\, S_7,\, S_8,\, S_{8,0},\, S_{8,1},\, S_9,\, S_{10,T},\, S_{10,F},\, S_{11,T},\, S_{11,F},\, S_{12,T},\, S_{12,F},\, S_{13},\, S_{14},\, S_{14,0},\, S_{14},\, S_{True},\, S_{False},\, \#,\, \$\} \cup$

  $\{\exists_{k_1,k_2,k_3} \mid k_1, k_2 \in \{T, F, \_\}, k_3 \in \{0,1\}\} \cup$

  $\{\exists_{k_1,k_2,k_3,M} \mid k_1, k_2 \in \{T, F, \_\}, k_3 \in \{0,1\}\} \cup$

  $\{\forall_{k_1,k_2,k_3} \mid k_1, k_2 \in \{T, F, \_\}, k_3 \in \{0,1\}\} \cup$

  $\{\forall_{k_1,k_2,k_3,M} \mid k_1, k_2 \in \{T, F, \_\}, k_3 \in \{0,1\}\}\ \}$;
- $\Sigma = \{S_{exit},\, S_{False},\, S_{True},\, \#,\, \$\ \}$;
- $R = R_0 \cup R_1 \cup R_2 \cup R_3 \cup R_4 \cup R_5$.

Because our grammar can decide any QSAT formula, it results that the Recognition Problem for this fixed grammar G is complete for PSPACE.

Now that we have an example of a PSPACE-complete grammar, we can easily give example of a virus for which detection complexity is PSPACE-complete. We consider our example of virus to be a program that

- knows our previously defined grammar rules
- contains a variable named 'w' that initially it's just the start symbol
- it has a function that generates the next value for variable 'w' by applying the known grammar rules
- generates a new program that is identical with current one and only the value for the word 'w' will be different, this new generated program is written to disk and executed

As we know from Cohen [Coh85], the virus detection problem is undecidable in general. This corresponds to the case of using unrestricted grammars. In 2003, Spinellis gave example of bounded length viruses for which the exact detection is NP-complete [Spi03]. Those viruses solve instances of the Satisfiability problem.

We gave an example of a bounded length virus for which the complexity detection is PSPACE-complete. For this complexity result we are taking in consideration an exact detection and not a heuristic one. A heuristic detection can detect this program in linear time, for example totally ignoring the value for the word 'w' but it will also have false positives.

Chapter 3

Anti emulation techniques used by malware packers

In this chapter we will present how malware packers used and continuously updated their anti-emulation techniques in order to evade detection.

We will present how 5 malware packers evolved for a cumulated period of time from 2009 to 2015 [LCG18]. At the end we will also present what anti-emulation techniques we found in recent malware packers from 2018.

Bredolab (2009-2010) is a packer that is close to the moment when malware packers started to be used as a service.

It started by using a specific interrupt instruction (INT 2E) as an anti-emulation technique. It also used rarely used instructions for a short period of time. Its main anti-emulation technique that kept updating it was the generation of exceptions and redirecting the execution from the previously registered exception handler. What it changed over time was the way of generating exceptions, it used 'hlt' instruction, 'int 3' instruction and various ways to write to a location of memory where current application didn't have write access: 'sub [0x401000], eax', 'xchg [0x401100], edx', 'and [0x401000], ecx'. This is an old packer and the techniques that it was using most probably will not be successful today as the emulators improved and some of these techniques could trigger suspicious behavior by their simple presence.

VIZ (2010-2014) started by checking the returned value of API functions and also checked the API function code to start with a specific assembly instruction. Its main anti-emulation technique was to check the content of various library files. What it did was to load a specific library in memory, like kernel32.dll, and then to check if specific fields from its executable header are between specific limits. It started by checking the value for Entry Point and then the value for SizeOfCode. This technique works because the emulated environment is not identical with a real operating system, those libraries don't have same properties as the ones from a complete operating system. Later, it started to compute an approximation for the library's size of code without reading the value directly from the header. This way forcing more work to update the emulator in order to fix that particular library that was checked.

VIZ 2 (2013-2015) started with a complicated way of checking the return value of API functions. It tested if a specific API function has a proper implementation by testing if it properly processes the input. For example at a moment in time it checked if ‘TF_IsCftmonRunning‘ function properly checks if a mutex exists in current system. It did that by creating that mutex before calling ’TF_IsCftmonRunning’ and checking if it returned the expected value. After a period of time their creators decided to use different anti-emulation techniques. Fist big change was to check the mouse movement. It was interesting that this technique lasted for almost 6 months without updating it. What it did was to check that the mouse cursor moves at least once in 60 seconds, but it doesn’t move in the first millisecond. After 6 months of using this mouse movement technique it started to use a similar technique as VIZ, the one of checking the content/integrity of library files. It used this new technique from June 2014 until December 2015. It was checking BaseRelocationTable size field from executable header and what it changed over time was the library name. What was interesting was that since August 2015 the library names that were used were sorted in alphabetical order, suggesting that at that moment its creators automated this anti-emulation technique to tell them what is the next library to be used after current one was somehow fixed in emulators.

UPA 1 (2013-2015) is a malware packer that started with the usage of rare instructions and rare API functions. Then as initial technique that it continuously updated was the usage of callbacks. It used TLS callbacks, RtlRegisterSecureMemoryCacheCallback and others. We consider the usage of callbacks a complicated technique because callbacks are not easy to be solved by an emulator but are also complicated to be found by the malware packer creators after previous callbacks can not be used any more. After a period of time they started to use a different technique: the usage of big loops. This technique works because an emulator can not execute a sample for minutes and usually an emulator doesn't execute code as fast as a real operating system. What they changed over time was the number of iterations and the way loops are integrated. Initially, loops were implemented as functions with the number of iterations as parameter, and later they integrated the loops code in the main code making it more complicated to identify the loop.

UPA 3 (2013-2015) is different from the previous packers in the anti-emulation technique that it is using and also by the that fact that it was exclusively used by Upatre downloader. Its main anti-emulation technique is interesting, it used Windows messages in order to stop the execution in an emulator. What it did was to create an application with interface, to put a button, a text box, a text area and then to send various messages and to check if it has the expected behavior. For example at one moment in time created a text area where a text was provided at initialization and then by sending EM_GETLINECOUNT message it was expecting to receive the number of lines for that specific text. It was interesting that its creators insisted in using only this type of anti-emulation technique, every time finding new Windows messages that were not properly handled by emulators.

We also analyzed 3 samples from 2018 to see what techniques are used recently. In first case we looked at a Kirts sample, that is also known as Worm.Phorpiex, and its particular malware packer was using big loops as an anti-emulation technique. It had multiple loops, that were easy to identify, one loop in particular had 3.714.383 iterations. What was interesting was that somewhere in the middle of the loop, at iteration 2.015.748 was initializing a variable with the address of kernel32.dll library and most likely was later checking if that variable was properly initialized. We assume that this particular check was to detect if some particular emulator identifies the loop and decides to pass over it without executing it, in that case that variable would not be properly initialized and later in code it could detect that the loop was not completely executed. Next sample that we analyzed is a Ursnif, that is a banker, in that case the malware packer that it was using was also using big loops as an anti-emulation technique, but in that case was not easy to identify from were a loop begins and when it ends. The last sample that we analyzed is Ramnit, that is also a banker, and in that case its malware packer was checking for the contents of library files, a technique similar with that VIZ packers were using. In particular that sample was checking ntdsapi.dll BaseReloc size. We observe how recent samples are using with success the same anti-emulation techniques as in previous years.

By giving previous examples we showed how anti-emulation techniques are used by malware packers and how are continuously updated in order to evade detection. We observed that usually at the beginning of a malware packer evolution, complex techniques are used, probably because its creators have more development time and ideas at the beginning, and later they decide to use simpler techniques that are easy for them to update and good enough to stop the emulators. Most of the time the malware packers will end up using big loops or the check of library files as anti-emulation technique. These two being the most used anti-emulation techniques in recent samples too.

Because the creators for malware packers have access to latest anti-malware detection, they can test their protection and get notified when an anti-malware program detects their files and this way update their code in order to evade the new added detection in a short period of time. By having access to current anti-malware detection (that can be considered the oracle), the detection for new samples that are protected by malware packers is undecidable.

Chapter 4

Using Hidden Markov Models to detect metamorphic code

We used Hidden Markov Models to test detection for Evol file infector samples [CL17]. Evol is known to be a highly metamorphic virus that evolves significantly across two generations.

In [Des08] HMMs were used to detect a polymorphic engine that implements the techniques present in Evol virus engine. The authors developed a polymorphic engine starting from analyzing the engine code used in Evol. In comparison to their research, we performed our training and testing on real samples that are infected with Evol virus.

Initially we tried to use a virus construction kit that emulated Evol engine but we were not satisfied with the generated output and we looked for real samples of Evol in order to test on real samples. We were lucky to find two Evol samples on vxheaven.org website [Evo] and those two samples were patient 0 samples. A patient 0 sample is a specially crafted sample that was created in order to spread that virus.

Starting from a patient 0 sample we executed it in a virtual machine in order to obtain infected samples, and we grouped the resulted samples by generation. For first generation we executed a patient 0 and kept only one sample that was the result of infection without using the mutation engine. Evol doesn't use its mutation engine for every infections so we wanted to keep for first generation only one infected sample that contains the file infector code but without any mutation. For second generation we executed the sample from first generation and collected only infected samples that were the result of a mutation. We repeated this step until we collected 100 samples for second generation. For third generation we chose a random sample from second generation, executed it in a virtual machine and collected only the samples that were the result of a mutation. We repeated this step until we collected 100 samples also for third generation. We continued with this steps until we collected 100 samples for ninth generation. We couldn't go further ninth generation because the samples from ninth generation could not infected other samples from current virtual machine that were the result of a mutation. That was most likely because the virus code increased significantly from generation to generation and it couldn't find new files that were able to contain its new mutated code.

After having samples separated by generation, we processed every sample in order to train and test Hidden Markov Models. Because Evol code is found at the Entry Point of infected file, it is easy to know from where it starts but it is not easy to know where it ends. That is why we decided to extract 6000 bytes from Entry Point to be sure that we extract only code that belongs to Evol for an infected sample. Those 6000 bytes contain only Evol code because the original virus code without any mutation is greater than 6000 bytes. After extracting a sequence of 6000 bytes we converted it to x86 assembly instructions and kept only the instruction name by ignoring its parameters. This way we obtain a list of instruction names. Next we encode each instruction name with a number to obtain a list of numbers that later can be used to train and test a Hidden Markov Model.

For learning Hidden Markov Models parameters we used unsupervised Baum-Welch algorithm and used the previously extracted sequences of numbers. To measure similarity of samples we compute log-likelihood.

For the experiments that we will present next it is important to start by presenting how much Evol code size increases from generation to generation. The initial virus code without any mutation has 6344 bytes. In ninth generation, after 8 mutations, the average virus code is 32468 bytes, an increase in size of more than 5 times than the original size.

The first experiment that we did was to test the assumption that it is important to extract and process only code that belongs to Evol virus. We knew that second generation virus code has a size between 7857 and 9538 bytes and we decided to extract 100 buffers of size 7000 and 100 buffers of size 10000 from those 100 samples from second generation. We know that extracting a buffer of size 7000 contains only virus code and a buffer of size 10000 also contains code from the host executable. We trained a HMM model on first 50 buffers of size 7000 and tested the model against the other 50 buffers of size 7000 and against those 100 buffers of size 10000. The results were as expected, the other 50 buffers of size 7000 were considered similar and the buffers of size 10000 were not considered similar. Knowing this result we decided to extract buffers of size 6000 for all generations to be sure that we process only code that belongs to Evol virus.

The next experiment that we did was to see how training a model on second generation or on ninth generation impacts detection for samples from the other generations. After extracting buffers of size 6000 from all generations we trained a model on second generation and tested it against generations 3 to 9 and we also trained a model on ninth generation and tested it against generation 2 to 8. In the first case, when we trained a model on second generation, all the other generations were considered similar. This was an expected result. But when we tested the model that was trained on ninth generation, only generation 7 and 8 were considered similar and generation 6 was partially considered similar. Our assumption for this result is that by extracting 6000 bytes from ninth generation we extracted only one fifth of virus code and the same 6000 bytes from previous generations contain more than one fifth and this way less functionality. Knowing this unexpected result we decided to process buffers of smaller sizes from previous generations and to test similarity with the same model that we trained on ninth generation on buffers of size 6000. We processed buffers of size 16, 32, 64, 128, 256, 512, 1000 and 3000. After testing similarity with new processed buffers we obtained results of similarity for all generations from 2 to 8 when processing a buffer of size 3000. From this experiment we conclude that if it is possible to separate samples by generation then it is better to train a model on first generations. In case that we don't have that separation, we obtained better results if we used a bigger buffer for training than the one used for testing.

Because initially we had two patient 0 samples we decided to train a model on second generation from first patient 0 and test against the generations 2 to 9 from the other patient 0. The result was of similarity for all tested generations meaning that this type of detection is generic enough to detect samples that belong to a similar virus strain that was not included in training.

Chapter 5

Conclusions

In first chapter we gave an example of a fixed grammar for which the recognition problem is PSPACE-complete and based on that grammar we gave an example of a bounded length virus for which detection is PSPACE-complete. This complexity result is for an exact detection that doesn't admit false positives.

In second chapter we showed how anti-emulation techniques are used in malware packers and how are constantly updated in order to evade detection. Anti-emulation techniques are still used in recent malware packers and this is one of the way the battle between anti-malware products and malware takes place.

In third chapter we used Hidden Markov Models to detect samples for Evol virus that is known to be highly metamorphic. As conclusions we want to mention that it is important to properly extract and process only code that belongs to the virus. In case that we can separate samples by generations it is better to train a model on first generations. In case that we have samples that are not separated by generation, training on a buffer that is bigger than the one used for testing can have better results. To use this type of detection first we have to identify samples that belong to a particular virus, to extract the virus code and then to train a model.

Bibliography

[Adl90] Leonard M Adleman. “An abstract theory of computer viruses (invited talk)”. In: *Proceedings on Advances in cryptology*. Springer-Verlag New York, Inc. 1990, pp. 354–374.

[BKM07] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. “A classification of viruses through recursion theorems”. In: *Computation and Logic in the Real World*. Springer, 2007, pp. 73–82.

[CL17] Doina Cosovan and Catalin Valeriu Lita. “Practical aspects related to using Hidden Markov Models for detecting metamorphic file infectors”. In: *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2017.

[Coh85] Fred Cohen. “Computer viruses”. PhD thesis. University of Southern California, 1985.

[Coh87] Fred Cohen. “Computer viruses: theory and experiments”. In: *Computers & security* 6.1 (1987), pp. 22–35.

[Des08] Priti Desai. “Towards an undetectable computer virus”. In: (2008).

[Evo] *Win32.Evol patient 0 samples.* [Accessed: 2017-06-25]. URL: http://vxheaven.org/vl.php?dir=Virus.Win32.Evol.

[Fil07] Eric Filiol. “Metamorphism, Formal Grammars and Undecidable Code Mutation”. In: *International Journal of Computer Science and Engineering* 1.2 (2007), pp. 0 –6. ISSN: 1307-6892. URL: http://waset.org/Publications?p=2.

[FS] F-Secure. *Threat description Brain.* URL: https://www.f-secure.com/v-descs/brain.shtml.

[LCG18] Cătălin Valeriu Liţă, Doina Cosovan, and Dragoş Gavriluţ. “Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in UPA packers”. In: *Journal of Computer Virology and Hacking Techniques* 14.2 (2018), pp. 107–126.

[Lit16] Catalin-Valeriu Lita. “On Complexity of the Detection Problem for Bounded Length Polymorphic Viruses”. In: *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2016 18th International Symposium on*. IEEE. 2016, pp. 371–378.

[Qoz99] Qozah. *Polymorphism and Grammars.* 1999. URL: http://vxheaven.org/lib/static/vdat/tupolgra.htm.

[Spi03] Diomidis Spinellis. “Reliable identification of bounded-length viruses is NP-complete”. In: *Information*

Theory, IEEE Transactions on 49.1 (2003), pp. 280–284.